Debugging Win95, Win NT, and Win CE Programs with the Microtek Pentium Emulator

Bruce Ableidinger
brucea@microtekintl.com
8/1/99

Introduction

The PowerPack and ITP emulators can now be used to trace, trigger, and do some debugging on a Windows 95 or NT device driver or application program. The SLD debugger is the controlling software for the emulator. Software version 2.9 NT Driver build xx is the version of SLD that added the necessary features for loading symbols and adjusting them to be able to bring up target source code (xx is 3 as of this document printing).

This product is primarily targeting NT or CE device driver developers who want to use the emulator trace to capture execution history prior to a problem. The emulator trigger can also be valuable in detecting and stopping the trace on a particular bug in the program. The trace Linked cursor mode will tie the trace execution history to the source code, either in Source or Mixed Source-Asm view.

This document primarily describes how to set up the emulator, load symbols, and adjust symbol addressing to provide source-level debugging and trace linkage to the device driver or application code.

Additions to SLD

PE symbol loading. Microsoft Visual C++ compiler/linker produces PE format files (PE is Portable Executable). SLD now loads modules, functions, and line number code symbols. This is adequate to provide source debugging (stepping, setting breakpoints) and linked cursors between the trace and source windows.

Also loaded are static and local variables. The latter includes fixed addresses for the statics and offset from stack frame for locals. Variable type information is not loaded, so the variable types are all set to unsigned long. A variable's type can be modified through casting. In the Variable Add dialog, the user can change from unsigned long to another type by using the syntax:

```
(<new type>)#<varname>
```

The variable type can also be cast to a pointer:

```
(<new type>*)#<varname>
```

Supported basic types are:

```
// 0x0001 - size:     0 bytes, void
// 0x0002 - size:     1 bytes, signed char
// 0x0003 - size:     1 bytes, unsigned char
// 0x0004 - size:     2 bytes, signed short int
// 0x0005 - size:     2 bytes, unsigned short int
// 0x0006 - size:     4 bytes, signed long
// 0x0007 - size:     4 bytes, unsigned long
// 0x0008 - size:     8 bytes, signed long long
// 0x0009 - size:     8 bytes, unsigned long long
// 0x000A - size:     4 bytes, float
// 0x000B - size:     8 bytes, double
```

```
// 0x000C - size:    10 bytes, long double
// 0x000D - size:    16 bytes, long long double
// 0x000E - size:     0 bytes, string
// 0x000F - size:     0 bytes, instruction address
// 0x0010 - size:     2 bytes, int
// 0x0011 - size:     2 bytes, unsigned
// 0x0012 - size:     2 bytes, unsigned int
// 0x0013 - size:     1 bytes, char
// 0x0014 - size:     4 bytes, long
// 0x0015 - size:     2 bytes, short
// 0x0016 - size:     2 bytes, unsigned short
// 0x0017 - size:     2 bytes, short int
// 0x0018 - size:     2 bytes, signed short
// 0x0019 - size:     8 bytes, BCD float
```

This type casting lasts only as long as the variable created in the Variable window. To permanently cast a variable (after symbols are loaded), you can use the vartype command, whose syntax is:

```
varType <#varname> [<type id>|<type name>[*]]
```

The type name is the right-most name in the list above. The <type id> is the left-most hex value. An example to convert a loaded symbol which is actually a string pointer is:

```
vartype #string1 char*
```

symsetsegment. This Shell command allows the symbol code segment to be changed to match the actual segment the driver or program is running under. It supports ldt selectors as well as gdt selectors.

symoffset. This Shell command adjusts loaded symbol addresses to match the actual load address of a device driver.

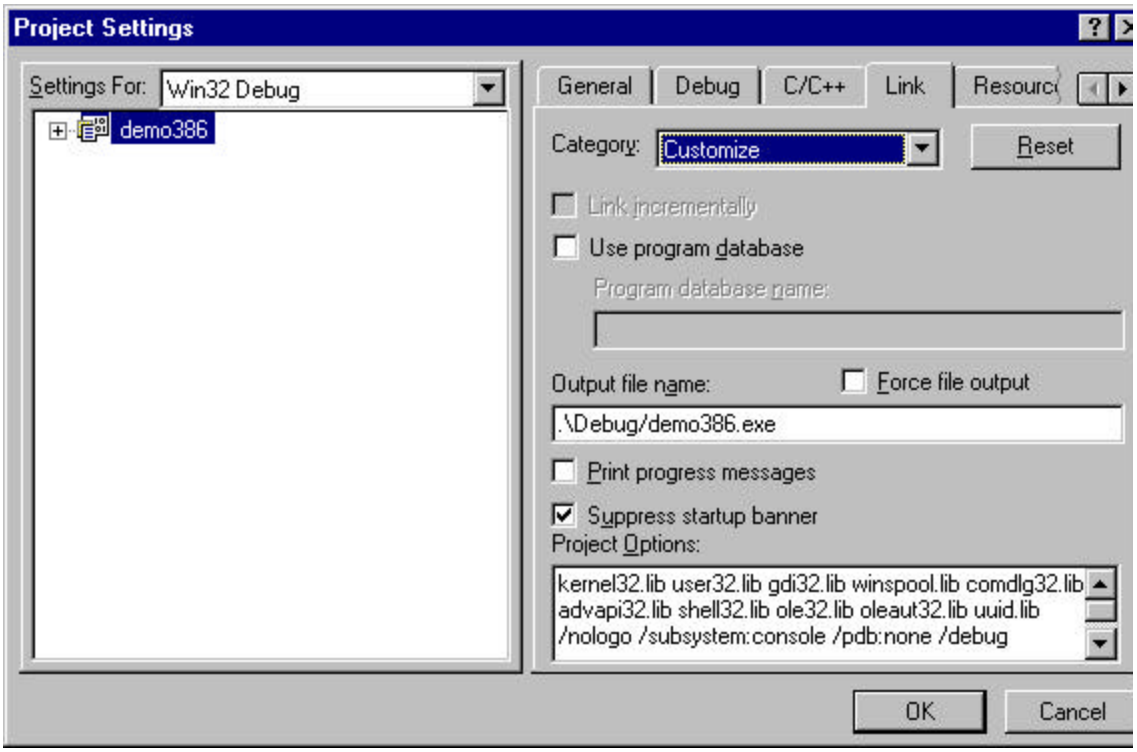
Steps to Get Emulator Synchronized to Target Program

Program Preparation. If you do not know where the program is loaded, or if the program load location changes, then it is recommended that you insert an ICEBREAK instruction at the beginning of the device driver or application code, such as the startup initialization.

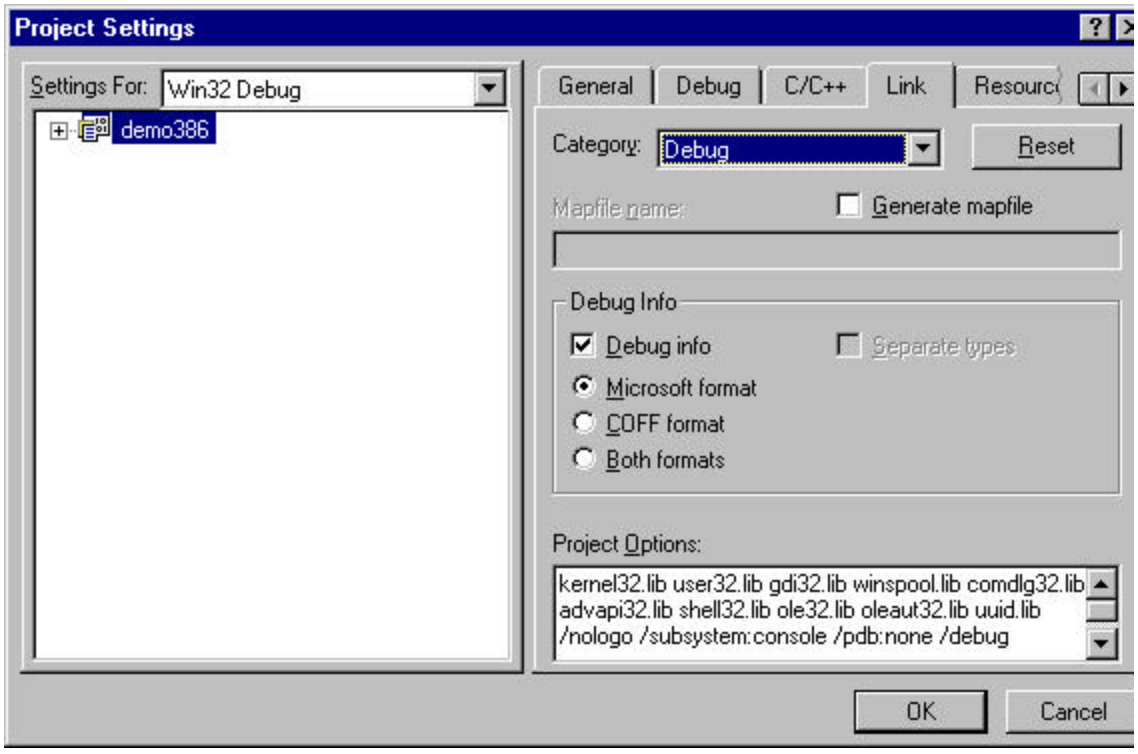
```
#define ICEBREAK _asm _emit 0xf1

main() {
   ICEBREAK
   <..>
}
```

Building the Program with PE symbols in the executable. Do NOT check “Use program database” in the Visual Studio Project/Settings/Link tab/Category Customize (you can use short-cut key Alt-F7 to bring up the dialog). If it is checked, symbols go into a separate .pdb file and the emulator will not be able to load PE symbols out of the .exe or .dll file.

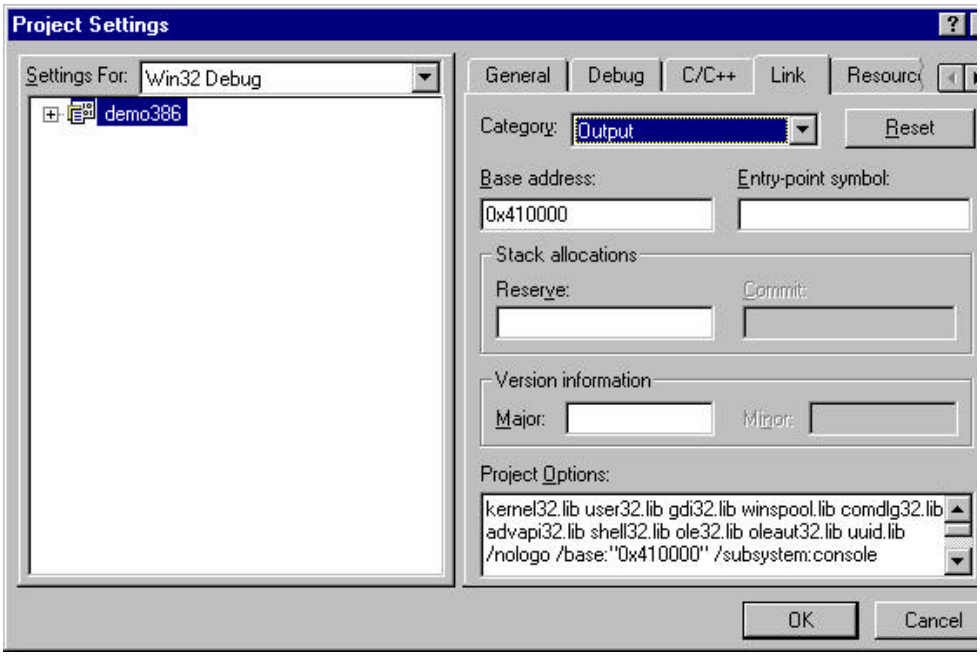


Building the Program with PE symbols in the executable. Check “Microsoft format” in the Visual Studio Project/Settings/Link tab/Category Debug (you can use short-cut key Alt-F7 to bring up the dialog). This tells the Linker to generate PE format files with embedded symbols.



The command line option for PE symbolic info is / pdb:none.

Relocating the Program. There may be a case where the program to be debugged is mapped at the same address space (e.g. 0x401000) as other applications running at the same time. Since the Pentium emulator uses BTMs to show execution trace even when the instruction cache is enabled, those linear address BTMs may “collide”, that is, overlap the other programs. Another reason is that debug registers compare to linear addresses, therefore to set a breakpoint on a unique program location, it needs to be in a unique linear address. To place the program to be debugged into a unique linear address space, use the following dialog box:



This is the Project Settings dialog, Link tab, Output Category. The Base Address field provides the entry for a different starting linear address for the .exe program.

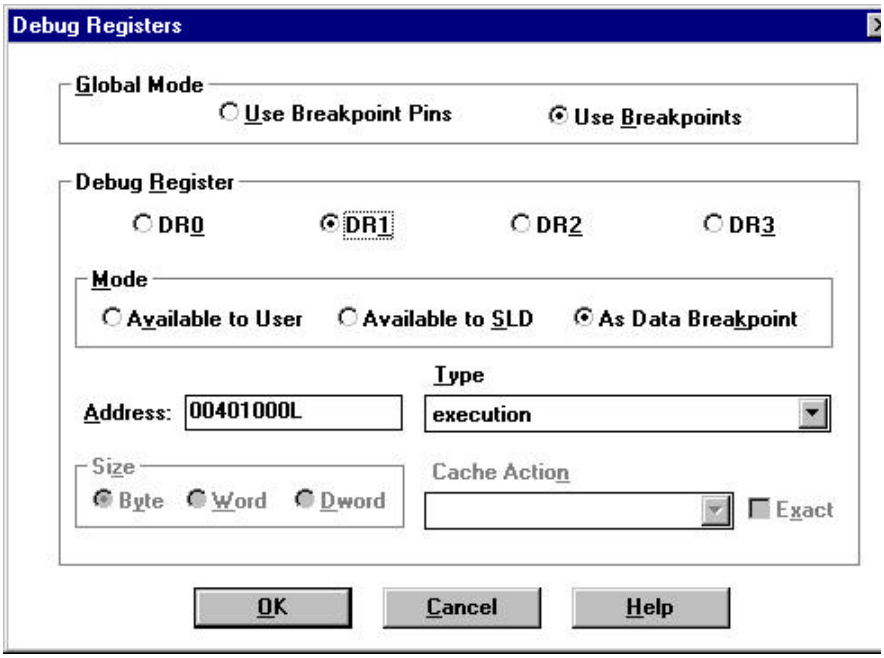
If a DLL is being debugged, its starting address can be modified with the **rebase** command line tool. Documentation on this command is in the InfoViewer, under the path Platform, SDK, and DDK Documentation/Platform SDK/Building Applications/Tools Guide/Tuning/Tuning the Working Set of your Application/Rebase. Or use Query Rebase to find and synchronize with it.

Halting the Target Program. If you do know the linear address where the program is loaded, you can program a debug register to break on the start of the program to be debugged. The command is:

```
dr 0 data x <linear_address> byte;
```

If debug register 0 is in use, you can use 1, 2, or 3. Note that the linear address should be typed with an ending “L” or “l”, like 0x401000L. This prevents it from being interpreted as a virtual or physical address.

You can also set a debug register from the CPU/Options/Debug Registers… dialog box, as follows:



Symbol Loading. Now, load program symbols. Example:

```
load “c:\demo386.exe” nocode nodemand;
```

Start the target

```
reset;go;
```

There may be a case where the BIOS uses or tests one of the debug registers, which by default causes the emulator to halt. If this occurs, allow that debug register to be assigned to the “user”; that is, not to be used by the emulator. For example,

```
dr 0 user;
```

This command can be added to the include.me file in the PowerPack directory, so that it is automatically disabled when the emulator is initialized.

Now wait for the target to boot the operating system. If the device driver is automatically loaded at boot time, then the ICEBREAK instruction will be hit and the emulator halted. This is the time where the symbols will be adjusted to match the target environment. If the target is running a program, then you should start the program so it can be loaded and the ICEBREAK hit.

Determining the Selector Value. When the ICEBREAK has been hit, bring up the CPU window and look at the CS register. If bit 2 is clear, the GDT is home of the selector. If bit 2 is set, then the program selector is from an LDT entry. The value in the LDTR register is the GDT entry pointing to the LDT table. The value in the CS register is the LDT selector value.

Note: the low-order 2 bits of the segment selector represent the RPL – requested privilege level. The CS register RPL field holds the CPL – current privilege level from the executing program. If the value is 3, the program is running in the outer Ring 3 level. The low-order 2 bits are not considered by the symbol table when looking up a match between the current CS and the stored segment value for symbols.

Aligning Symbols to GDT Selector. The PE symbols are loaded with a default GDT code value of 0x08 and a name of .text, and a GDT data value of 0x10 and a name of .data. To change this to match the program selector for a GDT selector, use the command:

symsetsegment <name> <sel_val> Example:

```
symsetsegment .text 0x18
symsetsegment .data 0x20
```

An example program running on Windows NT, when halted in the program, has a CS = 0x1B and DS, SS= 0x23. Since Bit2 = 0, this is a GDT selector. The symbol table initialization commands become:

```
symsetsegment .text 0x1B
symsetsegment .data 0x23
```

Aligning Symbols to LDT Selector. To change the default symbol selector to an LDT value after an ICEBREAK, use the command:

symsetsegment <name> <ldt:sel_val> Example:

```
symsetsegment .text 0xF0:0x137
```

The above is example of an application running on Windows 95; when halted in the program, has a CS = 0x137 and DS, SS= 0x13F. Since Bit2 = 1, this is an LDT selector, and LDTR = 0xF0.

Viewing the Current Symbol Selector Value. The name and value of each of the symbol selectors is obtained with the **getbase** command. Win95 example is:

```
getbase
// Base Name        LDT Selector     Seg/Selector     Base Address
// ============================================================
// *UND*            0000                              00000000
// .text            00F0             0137             00000000
// .data            0000             0010             00000000
// .bss             0000             0010             00000000
// *ABS*            0000             0010             00000000
// *COM*            0000             0010             00000000
```

Win NT example is:

```
getbase
// Base Name        LDT Selector     Seg/Selector     Base Address
// ============================================================
// *UND*            0000                              00000000
// .text            0000             001B             00000000
// .data            0000             0023             00000000
// .bss             0000             0023             00000000
// *ABS*            0000             0023             00000000
// *COM*            0000             0023             00000000
```

Adjusting Symbol Addresses. If the loaded addresses of the symbols do not match the actual loaded location of the driver or program, use the symoffset command to adjust. The command syntax is:

```
// symOffset [basename] [[+|-]<value> | <oldValue> <newValue>]
```

If you know the amount of offset between the loaded symbol addresses and the actual code load address, enter it with the command

```
symoffset .text +0x1000;
// Symbols Adjusted: 60, Lines Adjusted: 3573
// Overflowed Adjustment: 0, Underflowed Adjustment: 0
// Number of Affected Bases: 1
```

If you know the loaded address after the ICEBREAK has been hit you can use the <oldValue> <newValue syntax:

```
addressof #Dm_main#64
 // 0137:00401166..00401166
// assume the address at the ICEBREAK instruction is 0x801166
symoffset .text 0x401166 0x801166;
// Symbols Adjusted: 60, Lines Adjusted: 3573
// Overflowed Adjustment: 0, Underflowed Adjustment: 0
// Number of Affected Bases: 1
```

If the basename is left out, all loaded symbol addresses are modified.

```
symoffset
// Base Name        Offset Adjustment
// ================================================================
// *UND*            +0x0
// .text            +0x0
// .data            +0x0
// .bss             +0x0
// *ABS*            +0x0
// *COM*            +0x0
```

This is what it looks like after the `symoffset .text +0x100` command:

```
symoffset
// Base Name        Offset Adjustment
// ================================================================
// *UND*            +0x0
// .text            +0x100
// .data            +0x0
// .bss             +0x0
// *ABS*            +0x0
// *COM*            +0x0
```

Important note: the symoffset value is maintained in the powerpak.ini file and remains in affect between SLD sessions. If you want the offset disabled, you must issue a symoffset command with a value of the opposite sign of the value entered in order to cancel the current offset value saved. For example, if the .text value is +0x100, issue the command:

```
symoffset .text -0x100
```

Stepping Over ICEBREAK. When the ICEBREAK instruction is hit and the symbol base value of its gdt or ldt has been updated and symoffset value adjusted if needed, you will want to skip over this instruction. One way is to modify the register EIP with the Shell command.

```
reg eip;
//      EIP = 0x00401166
```

```
reg eip 0x401167;   // increment by 1
bkpt #Dm_func;    // set a breakpoint if you want to get control back
go;
```

You can also step over ICEBREAK from the Source window. You may want to set a breakpoint first to get control. You may also want to switch to View/Mixed Source and Asm to view each individual instruction. Then, click on the source or asm line after the ICEBREAK line. Go to the Run menu and click on Go From Cursor. This modifies the EIP and starts execution from the point of the cursor.

If your driver or program code is loaded at the same location every time, it may be worthwhile to remove the ICEBREAK instruction from your source code (and rebuild), then use a debug register or software breakpoint to stop the program where you want.

Disabling and Enabling Breakpoints. Sometimes it is useful to leave breakpoints programmed but to disable them while you are debugging and experimenting. To do this, bring up the Breakpoint window, select a breakpoint, and click on Disable. To make it active again, select it and click on Enable.

Source Window. To allow source files to be accessible by the Source window, be sure to enter the path or paths of the source files. This is done with the Options/Source Path… dialog.

Trace Capture. To get the most out of trace, be sure that the Trace/Trace Capture… dialog has the following parameters set: Capture Mode: Bus Cycles, Instruction Mode Assist: checked (Instruction Mode Assist means BTMs (Branch Trace Messages) are enabled to be generated by the target processors. The Trigger Position can be any value – Pre, Center, or Post.

Aligning Trace with Source. An important feature of the emulator is aligning the trace history of execution with source and optionally assembly code of the program that executed. To do this, bring up the Trace window. Make sure View/Instruction is set, then make sure View/Linked Cursor is checked. Now when you scroll trace to a new instruction, the Source window will scroll to the same source line or instruction (depending on the Source window view selected).

Other Useful Commands. There are other Shell commands that can be useful for debugging. Type “help <command_name> for the complete command syntax.

gdt Displays the current gdt entries

ldt cs Displays the ldt entry defined by the value in the cs register.

dt <selector> Displays information about this selector entry. It figures out if it is a gdt or ldt table entry and looks up the value and displays it.

displaysymbols Displays all the modules, line number ranges, and address ranges.

_test clock Displays the target clock speed.

_test vcc Displays the target voltage.

runaccess on | off When **on**, the emulator allows access to the target while the emulator is running. It actually stops the target briefly to allow the access, then starts it again. This mode is useful for viewing code in the Source or Memory windows without halting the emulator.

version Among other information, this command shows the software and firmware version currently running the emulator. It is useful when you need to know whether it is running the latest version.